Universiteit Gent - TELIN - DDCM
2024–2025

# Workshop: DDL 1

## 1 Introductie

In de vorige workshop hebben we kennis gemaakt met het databankbeheersysteem PostgreSQL. We zagen hoe dit databankbeheersysteem ons toelaat om relationele databanken aan te maken op een server en we leerden hoe we op verschillende manieren met deze databanken kunnen communiceren. In deze workshop gaan we dieper in op hoe we de verschillende basiscomponenten van zo'n relationele databank fysiek kunnen implementeren. Dit proces noemen we dan ook het fysiek ontwerp. In Sectie 2 introduceren we de basiscomponenten van een relationele databank en bouwen een eerste fysiek ontwerp van de `wielrennen` databank met behulp van instructies die gecategoriseerd worden onder de datadefinitietaal (DDL) van SQL[1]. Daarna geven wij, in Sectie 3, een korte introductie tot gebruikersrechten in PostgreSQL.

## 2 Basiscomponenten

In de vorige workshop hebben we op drie verschillende manieren een databank met naam `wielrennen` aangemaakt: door de `CREATE DATABASE` instructie uit te voeren in `psql`, door diezelfde instructie uit te voeren in de query tool van pgAdmin 4 en door de gebruikersinterface van pgAdmin 4 te gebruiken. Een lege databank is echter niet erg nuttig om als gebruiker mee aan de slag te gaan. Daarom worden er steeds een heleboel componenten, zoals tabellen, primaire en vreemde sleutels, functies... in zo'n relationele databank geïmplementeerd. Deze componenten maken het mogelijk om data gestructureerd, correct en volledig op te slaan in de databank.

[1] `https://www.postgresql.org/docs/current/ddl.html`

Typisch worden de verschillende componenten van een relationele databank schematisch voorgesteld in een relationeel databankschema. Het relationeel databankschema van de `wielrennen` databank, dat we doorheen de oefeningenlessen van dit vak als leidraad zullen gebruiken, is gegeven in de Appendix van dit document. In het vervolg van deze workshop (en ook in de workshop 'DDL 2') introduceren we de belangrijkste componenten van het relationeel databankmodel en gaan we dieper in op hoe ze in PostgreSQL geïmplementeerd kunnen worden. In deze workshop focussen we enkel op de basiscomponenten van het relationele model. Meer geavanceerde concepten komen later aan bod in de workshop 'DDL 2'.

## 2.1 Basisrelaties

De (basis)relatie (of tabel) vormt hét fundament van het relationeel databankmodel. In elke tabel kunnen data opgeslagen worden met betrekking tot een specifiek type entiteit (bv. wielrenners, wielerteams, wedstrijden...). Het relationeel databankschema van de `wielrennen` databank bestaat uit 6 tabellen, die elk worden voorgesteld door een rechthoek in het schema: `wielerteam`, `wielrenner`, `wedstrijd`, `route`, `rit` en `uitslag`. Deze tabellen moeten allemaal fysiek binnen de `wielrennen` databank geïmplementeerd worden. Het aanmaken van een tabel gebeurt in PostgreSQL met behulp van de `CREATE TABLE` instructie[2]. Binnen deze instructie definieer je de verschillende attributen (of kolommen) van de tabel, samen met het datatype van deze attributen. Het datatype van een attribuut[3] dwingt af welke soort waarden er voor dit attribuut opgeslagen kunnen worden (bv. tekstuele waarden, gehele getallen, datums...).
Als voorbeeld vind je hieronder de SQL-instructie waarmee we de tabel `wielerteam` fysiek kunnen aanmaken. Deze tabel bestaat uit vier attributen met bijhorend datatype: `naam` (`varchar`), `afkorting` (`char(3)`), `landcode` (`char(2)`) en `status` (`varchar`). De SQL-instructie voor de aanmaak van deze tabel is

```
CREATE TABLE wielerteam
(
    naam varchar,
    afkorting char(3),
    landcode char(2),
    status varchar
);
```

Na het uitvoeren van deze instructie in de query tool zien jullie normaal gezien in het browservenster van pgAdmin 4 (eventueel na refreshen) de tabel `wielerteam`

[2] `https://www.postgresql.org/docs/current/sql-createtable.html`
[3] `https://www.postgresql.org/docs/current/datatype.html`

verschijnen[4].

## 2.2 Primaire sleutels

Bij de aanmaak van de tabel `wielerteam` zijn we een essentiële component van het relationeel model vergeten: de primaire sleutel. Deze verzameling van attributen moet, per definitie, unieke waarden aannemen voor elke rij die in de tabel zal worden opgeslagen. Dit maakt het mogelijk om elke entiteit (of elke rij) die in deze tabel wordt opgeslagen uniek te identificeren en hiernaar eenduidig te verwijzen. Bovendien dient deze verzameling van attributen ook irreducibel te zijn. Dit betekent dat, wanneer er uit de verzameling een attribuut weggelaten zou worden, er geen garantie meer is dat iedere rij uit de tabel een unieke combinatie van waarden draagt voor deze attributen. Let op, elke tabel kan slechts één primaire sleutel hebben.
In een relationeel databankschema wordt een primaire sleutel steeds bovenaan een rechthoek weergegeven, samen met de afkorting 'PK', wat staat voor 'Primary Key'. Zo kunnen we afleiden uit het relationeel schema van de `wielrennen` databank dat `naam` de primaire sleutel is van de basisrelatie `wielerteam`. Om deze primaire sleutel toe te voegen aan de fysieke implementatie van de tabel, voeren we onderstaande instructie uit.

```
ALTER TABLE wielerteam
    ADD CONSTRAINT wielerteam_pkey PRIMARY KEY (naam);
```

Zoals je kan zien past deze instructie de tabel `wielerteam` aan door een primaire sleutel-beperking (met een vrij te kiezen naam, in dit geval `wielerteam_pkey`) op het attribuut `naam` toe te voegen.
Hierboven hebben we het aanmaken van een tabel en het toevoegen van een primaire sleutel aan deze tabel uitgevoerd door middel van twee instructies. Je kan, echter, de primaire sleutel ook direct toevoegen aan een tabel bij de aanmaak ervan. Aangezien we de tabel `wielerteam` en de bijhorende primaire sleutel hierboven al aanmaakten, is het noodzakelijk om deze tabel eerst te verwijderen vooraleer we hem opnieuw kunnen aanmaken. De `wielerteam` tabel verwijderen doe je door middel van onderstaande instructie.

```
DROP TABLE wielerteam;
```

Vervolgens maken we de tabel `wielerteam` en de primaire sleutel-beperking op attribuut `naam` opnieuw aan. Deze keer gebruiken we echter één instructie.

[4] Een overzicht van alle tabellen binnen de `wielrennen` databank vind je in het browservenster van pgAdmin 4, onder 'Databases > wielrennen > Schemas > public > Tables'.

```
CREATE TABLE wielerteam
(
    naam varchar,
    afkorting char(3),
    landcode char(2),
    status varchar,

    CONSTRAINT wielerteam_pkey PRIMARY KEY (naam)
);
```

Merk op dat het resultaat van bovenstaande instructie identiek is aan het resultaat van de vorige instructies.
Een laatste manier om de tabel `wielerteam` aan te maken is door middel van onderstaande verkorte notatie.

```
CREATE TABLE wielerteam
(
    naam varchar PRIMARY KEY,
    afkorting char(3),
    landcode char(2),
    status varchar
);
```

Deze instructie kan je enkel en alleen gebruiken als de primaire sleutel uit slechts één attribuut bestaat.

## 2.3 Vreemde sleutels

Naast basisrelaties en primaire sleutels bestaat het relationeel model uit een derde cruciale component: de vreemde sleutel. Een vreemde sleutel-beperking stelt een verwijzing voor van een verzameling van attributen in een basisrelatie naar een verzameling van attributen in een (al dan niet andere) basisrelatie en wordt gebruikt om relaties tussen rijen in (verschillende) tabellen weer te geven. Deze beperking dwingt af dat elke combinatie van attribuutwaarden die wordt toegevoegd onder de desbetreffende attributen van de vreemde sleutel in de refererende basisrelatie, ook aanwezig moet zijn in de gerefereerde basisrelatie. Deze voorwaarde wordt referentiële integriteit genoemd. Bovendien moeten de attributen in de gerefereerde basisrelatie uniciteit afdwingen, anders is het niet mogelijk om eenduidig te verwijzen naar een rij in een andere tabel.
Neem, als voorbeeld, de basisrelatie `wielrenner`. Het attribuut `teamnaam` verwijst, via een vreemde sleutel-beperking, naar het attribuut `naam` in de basisrelatie `wielerteam`. In deze laatste basisrelatie dwingt dit attribuut inderdaad uniciteit af, aangezien `naam` de primaire sleutel is van de tabel `wielerteam`. Daarnaast mag een

rij in de tabel `wielrenner` enkel waarden aannemen voor het attribuut `teamnaam` die ook voorkomen onder het attribuut `naam` van een rij in de gerefereerde basisrelatie `wielerteam`. Stel bijvoorbeeld dat je een wielrenner wil toevoegen aan de databank met als teamnaam 'Soudal Quick-Step'. In dit geval zorgt de vreemde sleutel-beperking ervoor dat je deze wielrenner enkel kan toevoegen als er reeds een team met naam 'Soudal Quick-Step' is toegevoegd aan de tabel `wielerteam`.
In een relationeel databankschema wordt een vreemde sleutel steeds aangeduid door de afkorting 'FK', wat staat voor 'Foreign Key'. Deze afkorting wordt, in het geval van meerdere vreemde sleutels per tabel, aangevuld met een index 'x'. Uit het relationeel databankschema kan je dus afleiden dat er een vreemde sleutelbeperking gedefinieerd moet worden op attribuut `teamnaam` in de tabel `wielrenner` die verwijst naar attribuut `naam` in de tabel `team`. De volledige instructie om de tabel `wielrenner` aan te maken vind je hieronder terug.

```
CREATE TABLE wielrenner
(
    naam varchar,
    landcode char(2),
    geboortedatum date,
    geboorteplaats varchar,
    gewicht double precision,
    lengte double precision,
    teamnaam varchar,

    CONSTRAINT wielrenner_pkey PRIMARY KEY (naam),

    CONSTRAINT wielrenner_wielerteam_fkey FOREIGN KEY (teamnaam)
        REFERENCES wielerteam (naam)
);
```

Ook voor vreemde sleutels die slechts uit één enkel attribuut bestaan is een verkorte notatie mogelijk die sterk lijkt op de verkorte notatie in het geval van primaire sleutels, geïntroduceerd in Sectie 2.2. Let wel op dat er expliciet meegegeven moet worden naar welke basisrelatie en naar welk attribuut de vreemde sleutel verwijst. Een voorbeeld hiervan wordt gegeven in de PostgreSQL documentatie met betrekking tot vreemde sleutels.
Tot slot is het belangrijk om even stil te staan bij de volgorde waarin basisrelaties aangemaakt moeten worden. In bovenstaand voorbeeld hebben we eerst de tabel `wielerteam` gedefinieerd, en vervolgens pas de tabel `wielrenner`. De omgekeerde volgorde zou namelijk niet mogelijk geweest zijn. Aangezien `wielrenner` een vreemde sleutel heeft die verwijst naar `wielerteam`, kan de tabel `wielrenner` pas worden aangemaakt nadat de tabel `wielerteam` is aangemaakt.

Denk eens na over de volgorde waarin je de basisrelaties van de `wielrennen` databank correct kan aanmaken. Zijn er meerdere volgordes mogelijk?

Implementeer alle geïdentificeerde basisrelaties, primaire sleutels en vreemde sleutels uit het relationeel schema van de `wielrennen` databank in de huidige `wielrennen` databank die draait op jouw lokale PostgreSQL cluster.

## 2.4 Overige beperkingen

Bovenstaande informatie stelt je in staat om een databank met basisrelaties, primaire sleutels en vreemde sleutels fysiek aan te maken. Deze componenten volstaan echter nog niet om alle functionaliteit die in het relationeel databankschema beschreven staat te implementeren. Het schema bevat immers ook een aantal extra beperkingen (of 'constraints'), die een resem voorwaarden opleggen waaraan (combinaties van) attribuutwaarden moeten voldoen. Om een volledige, 100% correct functionerende databank te bekomen moeten deze beperkingen dus ook op de een of andere manier in de fysieke databank geïmplementeerd worden. Merk op dat primaire en vreemde sleutels, net als datatypes, eigenlijk ook beperkingen zijn, aangezien ze ook bepaalde voorwaarden opleggen waaraan de waarden van attributen in een basisrelatie moeten voldoen. Daarnaast bestaan er echter nog andere soorten beperkingen, die in het vervolg van deze workshop worden besproken.

### 2.4.1 `UNIQUE`

Een `UNIQUE`-beperking vereist dat, voor een bepaalde verzameling van attributen in een basisrelatie, elke combinatie van waarden die in deze basisrelatie onder deze attributen wordt opgeslagen uniek moet zijn. Dit is bijvoorbeeld het geval voor de verzameling `{afkorting}` in de basisrelatie `wielerteam`. In tegenstelling tot primaire sleutels is het wel zo dat, wanneer enkel een `UNIQUE`-beperking wordt opgelegd aan een verzameling van attributen, er wordt toegestaan dat er `NULL`-waarden worden opgeslagen onder deze attributen. Zo'n waarde geeft aan dat de exacte waarde van een bepaald attribuut voor een bepaalde rij niet gekend is (zie ook Sectie 2.4.2). Het toevoegen van een `UNIQUE`-beperking (met naam `UC_relatie`) aan een reeds aangemaakte basisrelatie (met naam `relatie`) over de attributen $\{attr_1,...,attr_n\}$ gebeurt in PostgreSQL door middel van de volgende instructie.

```
ALTER TABLE relatie
    ADD CONSTRAINT UC_relatie UNIQUE (attr1,...,attrn);
```

Merk op dat een `UNIQUE`-beperking opnieuw rechtstreeks gedefinieerd kan worden bij aanmaak van een basisrelatie, hetzij met de verkorte notatie in het geval van een enkel attribuut, hetzij door toevoeging van een `CONSTRAINT`-definitie aan de `CREATE TABLE` instructie. Dit is vergelijkbaar met de aanmaak van primaire en vreemde sleutels.

Voeg alle geïdentificeerde `UNIQUE`-beperkingen uit het relationeel schema van de `wielrennen` databank toe aan de huidige `wielrennen` databank die draait op jouw lokale PostgreSQL cluster.

### 2.4.2 `NOT NULL`

Een `NOT NULL`-beperking wordt steeds gedefinieerd op een attribuut en dwingt af dat iedere rij voor dat attribuut wel degelijk een waarde moet bevatten. De waarde `NULL` wordt, met andere woorden, voor dat attribuut niet aanvaard en dit attribuut wordt dan ook aangeduid als 'niet-optioneel'. Toevoeging van een `NOT NULL`-beperking voor het attribuut `attr` in een reeds aangemaakte basisrelatie (met naam `relatie`) kan als volgt.

```
ALTER TABLE relatie
    ALTER COLUMN attr SET NOT NULL;
```

Deze instructie zorgt dus voor een aanpassing aan de definitie van de gegeven tabel en van een attribuut in deze tabel. Daarnaast kan je een `NOT NULL`-beperking definiëren bij aanmaak van een basisrelatie door `NOT NULL` te plaatsen na de definitie van het betreffende attribuut in de `CREATE TABLE` instructie.

**Belangrijk:** Een primaire sleutel is in essentie niets anders dan een verzameling van attributen waarop intern een `UNIQUE`-beperking is gedefinieerd en waarvoor, op elk van de afzonderlijke attributen in de verzameling, een `NOT NULL`-beperking is opgelegd. Er kunnen in eenzelfde basisrelatie meerdere verzamelingen van attributen zijn waarop zowel `UNIQUE`- als `NOT NULL`-beperkingen worden gedefinieerd. Slechts één hiervan kan echter gedefinieerd worden als primaire sleutel van de basisrelatie. De andere verzamelingen noemen we alternatieve sleutels.

Voeg alle geïdentificeerde `NOT NULL`-beperkingen uit het relationeel schema van de `wielrennen` databank toe aan de huidige `wielrennen` databank die draait op jouw lokale PostgreSQL cluster.

### 2.4.3 `CHECK`

Een laatste, veelgebruikte soort beperking is de `CHECK`-beperking. Dit soort beperkingen wordt gebruikt om af te dwingen dat een of meerdere waarden binnen een rij moeten voldoen aan een gegeven booleaanse expressie. Typisch wordt dit gebruikt om een restrictie te leggen op het domein (de toegestane waarden) van een attribuut (bv. status $\in$ {'World Tour', 'Pro Tour'}). Daarnaast kan dit ook gebruikt worden om waarden van verschillende attributen binnen dezelfde basisrelatie met elkaar te vergelijken (bv. status = 'Did Not Start' $\Rightarrow$ bonustijd is null).
Een `CHECK`-beperking kan dus bijvoorbeeld gebruikt worden om af te dwingen dat een wielerteam enkel de status 'World Tour' of 'Pro Tour' kan hebben. Deze beperking kan worden geïmplementeerd in de fysieke PostgreSQL databank met behulp van volgende instructie.

```
ALTER TABLE wielerteam
    ADD CHECK (status in ('World Tour', 'Pro Tour'));
```

Wanneer je later zal proberen om wielerteams in te voeren met een waarde voor het attribuut `status` die niet gelijk is aan één van bovenvermelde waarden, dan zal PostgreSQL omwille van deze beperking een foutmelding opwerpen.
Net zoals bij de andere beperkingen moeten `CHECK`-beperkingen niet noodzakelijk na aanmaak van een tabel worden toegevoegd, maar kunnen ze ook meteen bij aanmaak van een tabel worden gedefinieerd. Bovendien is ook hier de verkorte notatie mogelijk.

Voeg alle geïdentificeerde `CHECK`-beperkingen uit het relationeel schema van de `wielrennen` databank toe aan de huidige `wielrennen` databank die draait op jouw lokale PostgreSQL cluster.

### 2.4.4 Views, triggers en functies

Door middel van de bovenvermelde componenten kunnen nu quasi alle beperkingen die weergegeven worden in het relationele schema geïmplementeerd worden op het

fysieke niveau. Er bestaan echter nog twee andere soorten vereisten waarvoor meer geavanceerde componenten nodig zijn.
Allereerst heb je een component nodig waarmee eenvoudig afgeleide data voorgesteld kunnen worden. Dit zijn data die het resultaat zijn van een berekening waarin gebruik wordt gemaakt van andere data en die eenvoudig opgevraagd moeten kunnen worden. Daarom worden afgeleide data typisch voorgesteld door middel van views. Volgens het relationele schema van de `wielrennen` databank zijn er niet meteen afgeleide data die moeten worden opgeslagen. Je zou echter wel, als voorbeeld, een view kunnen definiëren waarin het aantal ritoverwinningen per wielrenner wordt bijgehouden.
Daarnaast zijn er ook nog beperkingen waarvoor geen van bovenvermelde componenten gebruikt kan worden om ze af te dwingen in een relationele databank. Een voorbeeld van zo'n beperking is dat een wielrenner op eenzelfde datum slechts aan 1 rit kan deelnemen. Dit soort beperkingen kan niet afgedwongen worden door middel van eenvoudige `CHECK`-beperking. De reden hiervoor is dat, wanneer je wil verifiëren of (nieuwe) data (een rij in de `uitslag` tabel) voldoet aan deze beperking, er ook data uit andere rijen, die zich reeds in de `uitslag` tabel bevinden, nodig zijn. Je hebt dus, met andere woorden, niet enkel data nodig van de rij die je momenteel aan het beschouwen bent. Om dergelijke beperkingen af te dwingen, heb je meer geavanceerde componenten, zoals functies en triggers, nodig.
Voorlopig is het nog niet nodig om te weten hoe je deze meer complexe beperkingen fysiek kan implementeren. In de workshop 'DDL 2' zullen we hier namelijk dieper op ingaan.

### 2.4.5 `serial`-datatype

Wanneer je in PostgreSQL een (artificiële) surrogaatsleutel (typisch een ID) wil implementeren (= een automatisch gegenereerd incrementerend getal) kan dit met het `serial`-datatype. Dit is eigenlijk geen écht datatype, maar het is een PostgreSQL-specifieke shortcut om aan te geven dat automatisch een unieke integer-waarde gegenereerd moet worden voor elke rij met dit attribuut[5]. Het onderliggende datatype van een '`serial`' attribuut zal dus `integer` zijn. Dit is belangrijk wanneer je later met een vreemde sleutel naar dit attribuut wil verwijzen. Het datatype van dit vreemde sleutel-attribuut moet dus ook `integer` zijn.

> Neem een backup van de `wielrennen` databank die draait op jouw lokale PostgreSQL cluster door uitvoering van het `pg_dump` commando. Bestudeer de inhoud van het backup-bestand nauwgezet en zorg ervoor dat je alles begrijpt.

[5] `https://www.postgresql.org/docs/current/datatype-numeric.html#DATATYPE-SERIAL`

# 3 Rollen en privileges

Als alles goed gegaan is, zou je op dit moment een fysiek geïmplementeerde `wielrennen` databank moeten hebben, die praktisch bruikbaar is en voldoet aan quasi alle vereisten. Deze databank zou vervolgens gebruikt kunnen worden door een internationale wielerorganisatie voor het beheer van hun wedstrijddata. Het is natuurlijk niet de bedoeling dat iedereen zomaar toegang heeft tot alle data die in de databank worden opgeslagen en al helemaal niet dat iedereen deze data kan manipuleren. Daarnaast zal er ook binnen de organisatie zelf bepaald moeten worden wie welke rechten krijgt op de databank. Zo zou er, bijvoorbeeld, door wielerfans geïnterageerd kunnen worden met de databank via een grafische interface. Zij zouden onder andere in staat moeten zijn om eenvoudig de wedstrijduitslagen van hun favoriete wielrenner op te vragen. Anderzijds willen we natuurlijk niet dat elke wielerfan (al dan niet per ongeluk) alle data uit een bepaalde basisrelatie, of nog erger, uit de volledige databank kan verwijderen. Naast wielerfans zijn er ook nog andere profielen die in contact komen met de databank, zoals de databankbeheerders. Aangezien deze personen instaan voor het beheer en het onderhoud van de databank, moeten zij natuurlijk toegang hebben tot de volledige databank en genoeg rechten hebben om alle beheerstaken zonder omwegen te kunnen uitvoeren.
Je merkt dus dat we een authenticatiemechanisme nodig hebben dat voor iedere connectie met de databank bijhoudt welk type gebruiker (rol) geconnecteerd is en wat dit type gebruiker mag doen. Bovendien willen we aan iedere rol specifieke acties en handelingen kunnen koppelen die deze rol mag uitvoeren. Gelukkig voorziet PostgreSQL een dergelijk authenticatiemechanisme. PostgreSQL werkt namelijk met zogenaamde ‘database roles’ of databankrollen. Een rol kan zowel een individuele gebruiker (bijvoorbeeld ‘Jan Janssens’) alsook een groep van gebruikers (bijvoorbeeld ‘admins’) voorstellen. Een rol bestaat uit een naam, een wachtwoord en een geassocieerde lijst aan rechten (of privileges) die een persoon die via deze rol is geconnecteerd kan uitvoeren op de databank. Bij de installatie van PostgreSQL wordt er automatisch één rol aangemaakt, namelijk de ‘postgres’ rol. Dit is ook hoe jij momenteel bent geconnecteerd met de databank, al is dit in de praktijk natuurlijk niet altijd gewenst.
Om een nieuwe rol met naam ‘user’ en wachtwoord ‘password’ aan te maken kan je onderstaande instructie uitvoeren.

```
CREATE ROLE user WITH LOGIN PASSWORD ‘password’;
```

Wie de gebruikersnaam en het geassocieerde wachtwoord van een rol kent, kan dus via deze rol inloggen op de gegeven databankserver. Welke acties een gebruiker onder een bepaalde rol vervolgens kan uitvoeren binnen het dbms, hangt echter af van de rechten die aan deze rol zijn toegekend. Rechten kunnen zowel worden toegekend op het niveau van een databank, van een schema of van een tabel. Enkele belangrijke acties waarvoor rechten kunnen worden toegekend zijn de volgende.

- `SELECT`: recht om data te lezen in een tabel.
- `INSERT`: recht om data toe te voegen aan een tabel.
- `UPDATE`: recht om data aan te passen in een tabel.
- `DELETE`: recht om data te verwijderen uit een tabel.
- `TRUNCATE`: recht om alle data te verwijderen uit een tabel (zonder invloed van de gedefinieerde beperkingen).
- `REFERENCES`: recht om een vreemde sleutel te definiëren in een tabel. Om een vreemde sleutel te kunnen definiëren, moet dit recht zowel worden toegekend voor de tabel waarin de vreemde sleutel wordt gedefinieerd, alsook voor de tabel waarnaar de vreemde sleutel verwijst.
- `TRIGGER`: recht om een trigger te definiëren op een bepaalde tabel.
- ...

Indien een rol een bepaalde tabel aanmaakt, worden alle bovenstaande rechten op deze tabel automatisch aan deze rol toegekend. Aangezien je gedurende deze workshop steeds hebt gewerkt onder de 'postgres' rol, heeft deze rol dus alle bovenstaande rechten op de tabellen die je eerder aanmaakte in de `wielrennen` databank. Nieuw aangemaakte rollen hebben standaard echter geen rechten.

Maak in jouw lokale PostgreSQL cluster een nieuwe rol met naam 'test' en wachtwoord 'testpw' aan. Maak vervolgens, in pgAdmin 4, een nieuwe connectie met naam 'test_connectie' naar deze cluster aan, waarbij je jezelf identificeert met de 'test' rol. Je hoeft je oude connectie hier niet voor te verwijderen. Probeer vervolgens via de 'test' rol de data te lezen die in de tabel `wielerteam` zijn opgeslagen. Dit doe je door rechts te klikken op de naam van de tabel, en dan 'View/Edit Data > All Rows' te selecteren.

Wanneer je als 'test' rol de inhoud van de `wielerteam` tabel probeert te bekijken, zou PostgreSQL normaal gezien een foutmelding moeten opwerpen. Dat is logisch aangezien de 'test' rol geen leesrechten heeft op de tabel `wielerteam`. We moeten dus eerst leesrechten toekennen aan de 'test' rol vooraleer iemand die is ingelogd onder deze rol ook effectief data kan lezen. Connecteer hiervoor eerst terug met jouw lokale PostgreSQL cluster via de 'postgres' rol. Ken dan aan de 'test' rol leesrechten toe op de `wielerteam` tabel door uitvoering van volgende instructie.

```
GRANT SELECT ON wielerteam TO test;
```

Met deze instructie ken je dus leesrechten (SELECT-rechten) toe op de wielerteam tabel aan de rol met naam ‘test’.

Ken leesrechten toe aan de ‘test’ rol op de wielerteam tabel. Ga nu opnieuw naar je ‘test_connectie’ connectie en probeer nogmaals de data uit de wielerteam tabel te lezen.

Normaal gezien zou je nu een lege tabel moeten zien (aangezien we nog geen data hebben toegevoegd aan deze tabel). Je merkt echter dat je, in tegenstelling tot jouw vorige poging, de structuur (de attributen en de datatypes) van de tabel wel degelijk kan bekijken.
Door middel van de GRANT instructie kan je dus een specifiek recht op een tabel toekennen aan een rol. Alhoewel we dit enkel hebben getest voor leesrechten (SELECT), is de GRANT instructie bruikbaar voor elk recht dat kan worden toegekend in PostgreSQL (SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER...). Als je echter op een snelle manier aan een bepaalde rol alle mogelijke rechten op een bepaalde tabel wil toekennen, kan je onderstaande instructie uitvoeren.

```
GRANT ALL PRIVILEGES ON table_name TO role_name;
```

Tot slot is het ook mogelijk om alle rechten op alle tabellen in een databankschema toe te kennen door uitvoering van de volgende instructie.

```
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA schema_name TO role_name;
```

Stel dat je iemand (een deel van) zijn rechten op een bepaalde tabel wil ontnemen, dan kan dit ook. Hiervoor kan de REVOKE instructie gebruikt worden.

```
REVOKE x ON table_name FROM role_name;
```

Hierbij staat x voor het recht dat je de opgegeven rol wil ontnemen.
Indien je een rol wil verwijderen kan dit door uitvoering van de volgende instructie.

```
DROP ROLE role_name;
```

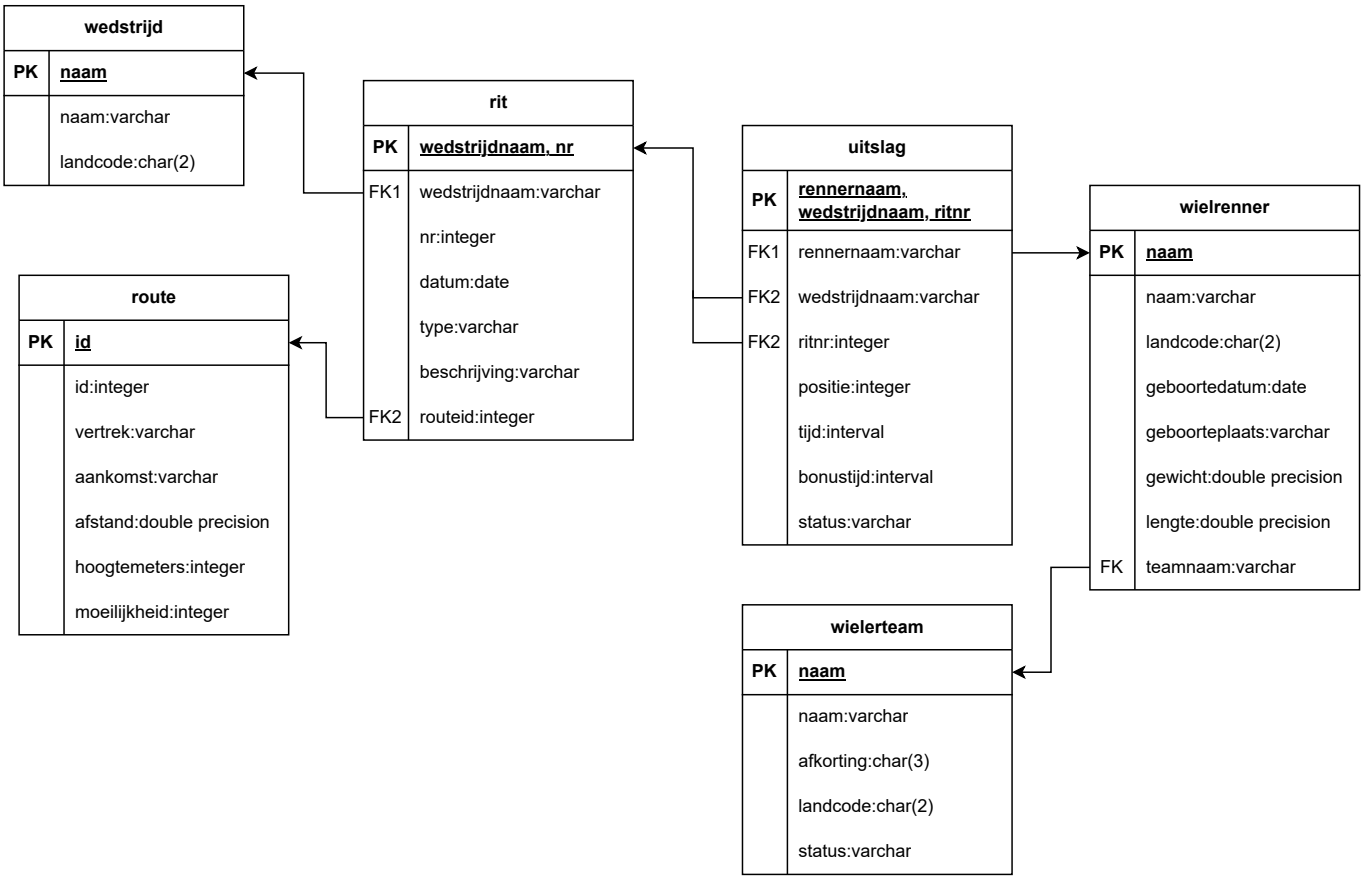
Let wel op dat je, vooraleer je een rol kan verwijderen, eerst alle rechten die toegekend zijn aan deze rol moet ontnemen.

Sluit de connectie ‘test_connectie’ af. Verwijder daarna vanuit de oorspronkelijk connectie de rol met naam ‘test’.

Meer informatie in verband rollen en privileges kan je terugvinden op https://postgresql.org/docs/current/user-manag.html.

## Appendix: Relationeel schema `wielrennen` databank

In onderstaande figuur vind je het relationeel databankschema van de `wielrennen` databank. Hierbij wordt iedere basisrelatie weergegeven door een rechthoek, die bovendien een oplijsting van alle attributen met bijhorende datatypes bevat. Daarnaast worden de attributen die behoren tot de primaire sleutel (PK) bovenaan weergegeven, en worden vreemde sleutels (FKx) voorgesteld door een pijl tussen de betreffende attribuutverzamelingen. Alle extra beperkingen die niet kunnen worden weergegeven in dit schema, worden hieronder opgelijst.



Extra beperkingen

- `wielerteam`:
  - uniek: {afkorting}
  - check: status ∈ {‘World Tour’, ‘Pro Tour’}
- `wielrenner`:
  - optioneel: geboorteplaats, gewicht, lengte
  - check: gewicht > 0
  - check: lengte > 0
- `route`:
  - check: afstand > 0
  - check: hoogtemeters ≥ 0
  - check: moeilijkheid ∈ [1,5]
- `rit`:
  - check: nr ≥ 1
  - check: type ∈ {‘Road Race’, ‘Individual Time Trial’, ‘Team Time Trial’}
- `uitslag`:
  - optioneel: positie, tijd, bonustijd
  - check: positie > 0
  - check: tijd > 0
  - check: bonustijd ≥ 0
  - check: status ∈ {‘Did Finish’, ‘Did Not Finish’,‘Did Not Start’, ‘Over Time Limit’, ‘Disqualified’}
  - check: status = ‘Did Finish’ ⇒ positie is not null ∧ tijd is not null ∧ bonustijd is not null
  - check: status ≠ ‘Did Finish’ ⇒ positie is null
  - check: status ∈ {‘Did Not Finish’, ‘Did Not Start’} ⇒ tijd is null
  - check: status = ‘Did Not Start’ ⇒ bonustijd is null
  - trigger: een wielrenner kan op eenzelfde datum slechts aan 1 rit deelnemen
  - trigger: indien de uitslagstatus van een renner in een rit niet gelijk is aan ‘Did Finish’, kan hij/zij niet meer deelnemen aan ritten in dezelfde wedstrijd met een hoger volgnummer